Whatever happened to meta-programming? (Invited talk)

Gallagher, John Patrick

Published in:
AGP-2002, Madrid, 17-20 September 2002

Publication date:
2002

Document Version
Publisher's PDF, also known as Version of record

Citation for published version (APA):
Gallagher, J. P. (2002). Whatever happened to meta-programming? (Invited talk). In *AGP-2002, Madrid, 17-20 September 2002* Universidad Politécnica de Madrid.

Whatever Happened to Meta-Programming?

John Gallagher
University of Bristol

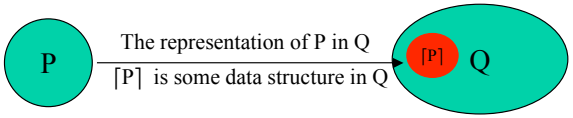
Youthful Expectations

- 10-15 years ago
  - Meta-interpreters + partial evaluation
  - Meta-interpreters "for real" -- to realise language implementation and extensions
  - Compiling control, clever *and* efficient program control
  - Self-applicable partial evaluation for generating compilers and compiler-compilers
  - Inventing abstract machines
  - Reflection (demo, holds, etc) for knowledge management and problem solving
  - Conferences for meta-programming (META, Programs as Data,..)
- Progress and continued activity, but…..

Middle-age realism

- The dreams live on - the problems are becoming better understood
  - how to express semantics
  - how to make more powerful partial evaluators
  - the essential role of static analysis
  - how to design good object program representation

Object Language and Meta Language



Object program P written in language L

Meta program Q written in language M (possibly L=M)

The real target for tools

- We usually talk of developing *language-specific* tools, e.g.
  - an analyser for C
  - a transformer for Prolog
  - a verifier for VHDL
- Each tool is complex and hard to develop
- We should analyse, transform, prove etc. in the meta-language
- One meta-language can serve many object languages

Language Semantics as a Parameter

- A systematic method for developing object language processors
- Explicit object language definitions, written down in a meta-language
  - syntax and semantics definitions
- Construct language independent tools, parameterised by semantics

One program executing another

- An interpreter for language L, Int_L
- Takes a (representation of) a program in L and some input data, and computes some output data

  $Int_L : Prog_L \times Input \rightarrow Output$
- The familiar partial evaluation equations (Futamura projections) show that we can "compile" a program

  $PE(Int_L, p)$ = "p compiled into the language of Int_L"

One more generalisation

- A universal interpreter U
- Takes a language definition (syntax and semantics) as parameter
- Parses and executes a program according to the given syntax and semantics

  $U : LangDefs \times Progs \times Input \rightarrow Output$
- PE wrt a fixed language definition

  $PE(U, Def_L) = Int_L$

Object language definition

- Syntax
  - abstract syntax = terms, parse trees
  - utilities (tokenizers and parsers) to convert concrete programs (e.g. strings, flowcharts) to abstract syntax
- Semantics
  - transitions
  - derivations
  - abstract machine states
  - denotations

Transition Systems

- Operational semantics
- Define abstract machine
  - Computation states
  - Transitions - binary relation on states
  - Derivations
- Transitions can be Big-step or Small-step

Big-step vs. small-step

- Big-step
  - the total effect of each program construct is expressed
  - initial state $\Rightarrow$ final state
- Small-step
  - the immediate effect of each statement is expressed
  - initial state $\Rightarrow$ next state

Small-step semantics

- Operational semantics (small-step or structural)
  - a set of computation states Q (configurations) including terminal states F
  - a set of transitions - a relation $\Rightarrow$ in $Q \times Q$
  - each transition describes a basic computation step
  - derivations (computations)
    - $q_0 \Rightarrow q_1 \Rightarrow q_2 \Rightarrow \ldots$
    - terminating derivation if $q_n \in F$

Big-step semantics

- Operational semantics (big-step or natural)
  - a set of computation states Q (configurations) including terminal states F
  - a set of transitions - a relation $\Rightarrow$ in Q × Q
  - each transition describes a complete computation for some program construct
  - derivations and transitions are not distinguished

Interpreter for small-step semantics

- a predicate transition(S1,S2) to represent S1 $\Rightarrow$ S2
- a predicate derivation(D) where D=[q0,q1,q2,…]

```
derivation([Q]) :- terminal(Q).
derivation([Q1,Q2|Qs]) :-
      transition(Q1,Q2), derivation([Q2|Qs]).
```
← whole derivation sequence is observed

```
derivation(Q,Q) :- terminal(Q).
derivation(Q1, Q) :-
      transition(Q1,Q2), derivation(Q2,Q).
```
← only initial and final states are observed

```
derivation(Q) :- terminal(Q).
derivation(Q1) :-
      transition(Q1,Q2), derivation(Q2).
```
← only current state is observed

Example - simple imperative language

(Peralta, Gallagher, Saglam, SAS'98, Peralta, PhD 2000)

- Computation state is a pair <Code, Store>
- In a terminal state, the Code is empty (ε)

<x := expr, t>	$\Rightarrow$ < ε, t[x/val(expr,t)] >
<if b then s1 else s2, t>	$\Rightarrow$ < s1, t > *if val(b,t) = true*
<if b then s1 else s2, t>	$\Rightarrow$ < s2, t > *if val(b,t) = false*
<s1;s2, t >	$\Rightarrow$ < s11; s2, t1 > *if <s1, t> $\Rightarrow$ <s11, t1> (s11 $\neq$ ε)*
<s1;s2, t >	$\Rightarrow$ < s2, t1 > *if <s1, t> $\Rightarrow$ <s11, t1> (s11 = ε)*
<while b do s, t>	$\Rightarrow$ <if b then (s;while b do s) else skip, t>
<skip, t>	$\Rightarrow$ < ε, t>

Syntax representation

x := expr	assign(X, Expr)
if b then s1 else s2	ifte(B,S1,S2)
s1;s2	compose(S1,S2)
while b do s	while(B,S)
skip	skip

```
i := 2;
j := 0;
while (n*n > 1) do
  {if (n*n = 2) then
    i := i+4;
  else
    i := i+2; j := j+1;
  }
```

```
compose(assign(i,2),
 compose(assign(j,0),
  while(n*n > 1,
        ifte(n*n=2,
          assign(i, i+4),
          compose(assign(i,i+2),
            assign(j,j+1))
        )
  )
)
```

Representation of transitions

Represent configuration <stmt, t> as c(Stmt, T)

```
transition(c(assign(X, Expr), T), c(empty, T1)) :-
                        val(Expr, T, V),
                        update(T,T1,X,V).
transition(c(ifte(B,S1,S2),T), c(S1, T1)) :-
                        val(Expr, T, true).
transition(c(ifte(B,S1,S2),T), c(S2, T1)) :-
                        val(Expr, T, false).
transition(c(while(B,S),T), c(compose(ifte(B,while(B,S),skip)), T)).
transition(c(compose(S1,S2), T), c(compose(S11,S2), T1)) :-
                        transition(c(S1,T), c(S11,T1)), S11 ≠ empty.
transition(c(compose(S1,S2), T), c(S2, T1)) :-
                        transition(c(S1,T), c(S11,T1)), S11 = empty.
```


Program Specialization

- PE(P,$input_1$) = P_{input1}
  - P($input_1$, $input_2$) = P_{input1}($input_2$)
- when applied to language interpreters, can be regarded as generalised compilation
  - PE(Int, P) = Int_P
  - Int_P is a translation of P into the language of Int
- self application yields compiler
  - PE(PE,Int) = PE_{Int}, and PE_{Int}(P) = Int_P

Specialization of Transition Semantics

```
derivation(I,J,N) :-
    d1(I,J,N).
d1(I,J,N) :-
    d2(J,N).
d2(J,N) :-
    d3(0,2,N).
d3(I,J,N) :-
    {Y = N*N},
     gt_test1(Y,R)
     d4(I,J,N,R).
```

```
d4(I,J,N,false).
d4(I,J,N,true) :-
    d5(I,J,N).
d5(I,J,N) :-
    {Y = N*N},
     eq_test1(Y,R),
     d6(I,J,N,R).
d6(I,J,N,true) :-
     d7(I,J,N).
d6(I,J,N,false) :-
    d8(I,J,N).
```

```
d7(I,J,N) :-
    {I1 = I+4},
    d3(I1,J,N).
d8(I,J,N) :-
    {I1 = I+2},
    d9(I1,J,N).
d9(I,J,N) :-
    {J1 = J+1},
    d3(I,J1,N).
```

Specialization of derivation semantics w.r.t. example program

Analysing at Meta-Level

The program above was submitted to a convex-hull analyser for CLP.
Analysis returns a set of linear constraints associated to each predicate call. E.g. (partial result).

```
d7_query(I,J,N) :- J >= 0, J-0.5*I =< -1.0
d9_query(I,J,N) :- J >= 0, J- 0.5*I =< -2.0
```

Each predicate d1, d2, etc. corresponds to an imperative program point. Thus the results can be interpreted back to the object level easily.

Big-step semantics

- The total effect of a complete language construct is specified
- Often specified in compositional style
- <Statement, Init, Final>

$$\frac{}{\langle x := expr, t, t[x/val(x,t)] \rangle} \qquad \frac{\langle s1, t, t1\rangle \ \langle s2, t1, t2\rangle}{\langle s1 ; s2, t, t2\rangle}$$

$$\frac{val(b,t) = true, \langle s1, t, t1\rangle}{\langle \text{if } b \text{ then } s1 \text{ else } s2, t, t1\rangle} \qquad \frac{\langle \text{if } b \text{ then } (s;\text{while } b \text{ do } s) \text{ else skip}, t, t1\rangle}{\langle \text{while } b \text{ do } s, t, t1\rangle}$$

$$\frac{val(b,t) = false, \langle s2, t, t1\rangle}{\langle \text{if } b \text{ then } s1 \text{ else } s2, t, t1\rangle}$$

Big-step interpreter

- The big-step transition relation functions also as a derivation relation.
- transition(S, Init, Final) can be specialized w.r.t. a given program S.
- The resulting program differs from the small-step semantics
  - intermediate states are visible.
  - program is not tail recursive

Specialized Big-step interpreter

```
i := 2;
j := 0;
while (n*n > 1) do
  {if (n*n = 2) then
    i := i+4;
  else
    i := i+2; j := j+1;
  }
```

```
d(I,J,N, I',J',N') :-
   d1(I,2,I1),
   d2(J,0,J1),
   d3(I1,J1,N,I',J',N'). % while
etc.
```

More complex analysis?
Variables are duplicated
Answers must be propagated during analysis

Labelled Transition Semantics

- A compositional style of small-step semantics
- Used extensively for process algebras
- Process language with constructs a.P, P1+P2, P|Q, P\L, P[f] (Milner)
- (Labelled) transitions, and derivations, are defined similarly to the imperative language
- Specialized meta-programs have been "model checked" using static analysis techniques (e.g. Leuschel et al.)

$$\frac{}{a.P \Rightarrow^a P} \qquad \frac{P \Rightarrow^a P1}{P|Q \Rightarrow^a P1|Q} \qquad \frac{Q \Rightarrow^a Q1}{P|Q \Rightarrow^a P|Q1} \qquad \text{etc.}$$

"Low-level" meta-language

- We can use a low-level meta-language to represent the semantics of a high-level object-language
- E.g. proof mechanism for full first order logic realised by Horn clause program
- Search space of prover pruned using analysis of Horn clause meta-program.

Requirements of meta-language

- convenient representation of syntax
  - important only for readability
- representation of transitions
  - relational style fits better with traditional style of semantic specification
  - non-determinism
- fixpoint semantics
  - framework for abstract interpretation

Static Analysis of Term Structure

Aim of set-based analysis (a.k.a. type inference- to find a safe approximation of the set of values that can appear at a given program point (work goes back to [Reynolds, 1968])

```
q([],X,X).
q([c(X1)|Y],Acc,X) :-
      integer(X1), q(Y,c(X1,Acc),X).
q([d(X1)|Y],Acc,X) :-
       integer(X1), q(Y,d(X1,Acc),X).
p(X,Y) :- q(X,0,Y).
```

S_Y ::= 0 | c(Int, S_Y) | d(Int, S_Y)

(S_Y is an infinite regular set of terms)

Uses of Set-based Analysis

For imperative program analysis the environment stack is modelled as a component of the state.

The stack can be unbounded in the presence of recursive procedures (hence, stack must be approximated)

Over approximation means that control is undefined at procedure exit.

Details - Gallagher-Peralta (HOSC, 2001)

Current and Future Work

- Using set-based analysis of meta-programs
- Verification experiments
  - infinite state model checking [Charatonik & Podelski]
  - cryptographic protocols [Abadi & Blanchet]
    - using meta-language representation of protocol
  - Shape analysis in imperative language

Related Research

- The Boyer-Moore prover (ACL2) (and other provers) is essentially a meta-level approach
  - First, systematically translate the object program/system to a LISP-like functional program
  - Verify the functional program
  - Interpret the results back to the object program
- The Supercompiler project (Turchin et al.)
  - target Java - execute Java via meta-level interpreter, and optimise at that level

Powerful tools for the meta-language

- Generally, analysis, proof, and transformation tools are better developed for declarative languages.
- Transport these advantages to other languages, by using declarative languages as a meta-language

Some day….

- Programming will cease to be an error-prone and time-consuming handcraft
- Most programs will be generated, verified by other programs
- Meta-programming is one of the unifying principles of computing
- Systematic meta-programming is key